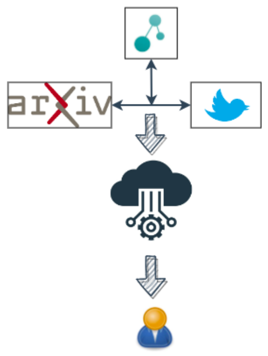
CSCI 5253

DATA CENTER SCALE COMPUTING

Project Report

ResearchDex



Submitted By

Nitin Kumar
Rohan Das
Tuhina Tripathi

TABLE OF CONTENTS

1. OVERVIEW
2. PROJECT GOALS
3. COMPONENTS
    a. Software and Hardware components
    b. Architecture Diagram
    c. Interaction of the components
4. DEBUGGING AND TESTING
    a. Debugging
    b. Testing
5. CAPABILITIES AND LIMITATIONS
    a. Capabilities
    b. Limitations

Overview

Today, given the pace at which research is advancing, it can be challenging to keep up with the latest developments. In rapidly advancing fields of Computer Science such as Machine Learning, Natural Language Processing, Robotics, Cybersecurity, the number of publications is growing exponentially. As a graduate student, one can sometimes feel overwhelmed with the information overload, and struggle to keep up with the new advances.

One way to stay up-to-date is to regularly read research journals and articles in one's field. However, it is often cumbersome to search for papers especially when one's interests span across a breadth of topics. Moreover, good papers often seem to slip throught the cracks.

We wanted to work on a solution that could help researchers stay on top of the literature in their area of interest. This motivated us to build a tool that aggregates the latest research papers in an area and presents it to users along with a link to the publication.

Project Goals

Our solution to this problem was a Slack Application called ‘Research Dex’ that can help researchers stay informed about the latest research. ‘Research Dex’ has a chatbot-like interface and users can use Slack’s Slash commands to interact with the tool. It gives users the following functionalities:

- **Enter topics of interest** which are associated with the user and are saved in the application’s database .
- **Update** the saved topics of interest.
- **Search** within the predetermined preferences by using a simple search command.
- **Search for random topics** by passing them as arguments to the search commands.

We built and hosted our backend services on Google Cloud Platform. These services are responsible for aggregating search results across multiple research areas and eventually create personalized search results for individual users. For the project, we integrated papers posted to ‘ArXiv’ but this can be easily extended to include other data sources such as Semantic Scholar and ACL Anthology.

Components

The Google Cloud Platform suite of services were used to build and host the server side components of the application. It provided us with a range of services for compute, storage and application development.

1. **Backend - Google Cloud Functions:** We employed a microservices architecture for the application. Most of the server side computations are going to run for a very short period of time, but with scale with expect a huge number of such computations to take place. Naturally Cloud Functions are a good fit for such microservices. Cloud Functions are event driven, stateless and scale automatically which perfectly fits the needs of our application.

2. **Storage - Datastore:** We used the NoSQL Google Datastore for storage as it's highly scalable and uses redundancy to minimize impact from points of failure, thus resulting in high availability of reads and writes.

3. **Message Queue - Pub/Sub:** Pub/Sub pattern was used to handle communication between different components thus allowing for decoupling. This is also a great choice for near real-time communication between components given it's push based delivery mechanism. This eliminates the need for any polling to check for messages in queues and reduces the delivery latency of the application. Pub/Sub can also easily scale with an increase in the number of user requests.

4. **Cron Job Scheduler - Cloud Scheduler:** We used the Cloud scheduler which is a fully managed cron job service that allows us to schedule repetitive tasks or tasks that need to be performed at a certain interval in an automated manner.

5. **API Interface - API Gateway:** Google API Gateway was used to expose REST endpoints for the Slack based user-interface. Given its seamless integration with Cloud Functions it was a natural choice.

6. **Message Marshalling:** We push JSON messages on to the Pub/Sub queues, however, Pub/Sub expects messages to be encoded into a byte string. Hence we employ message marshalling to handle message transformations between these two formats.

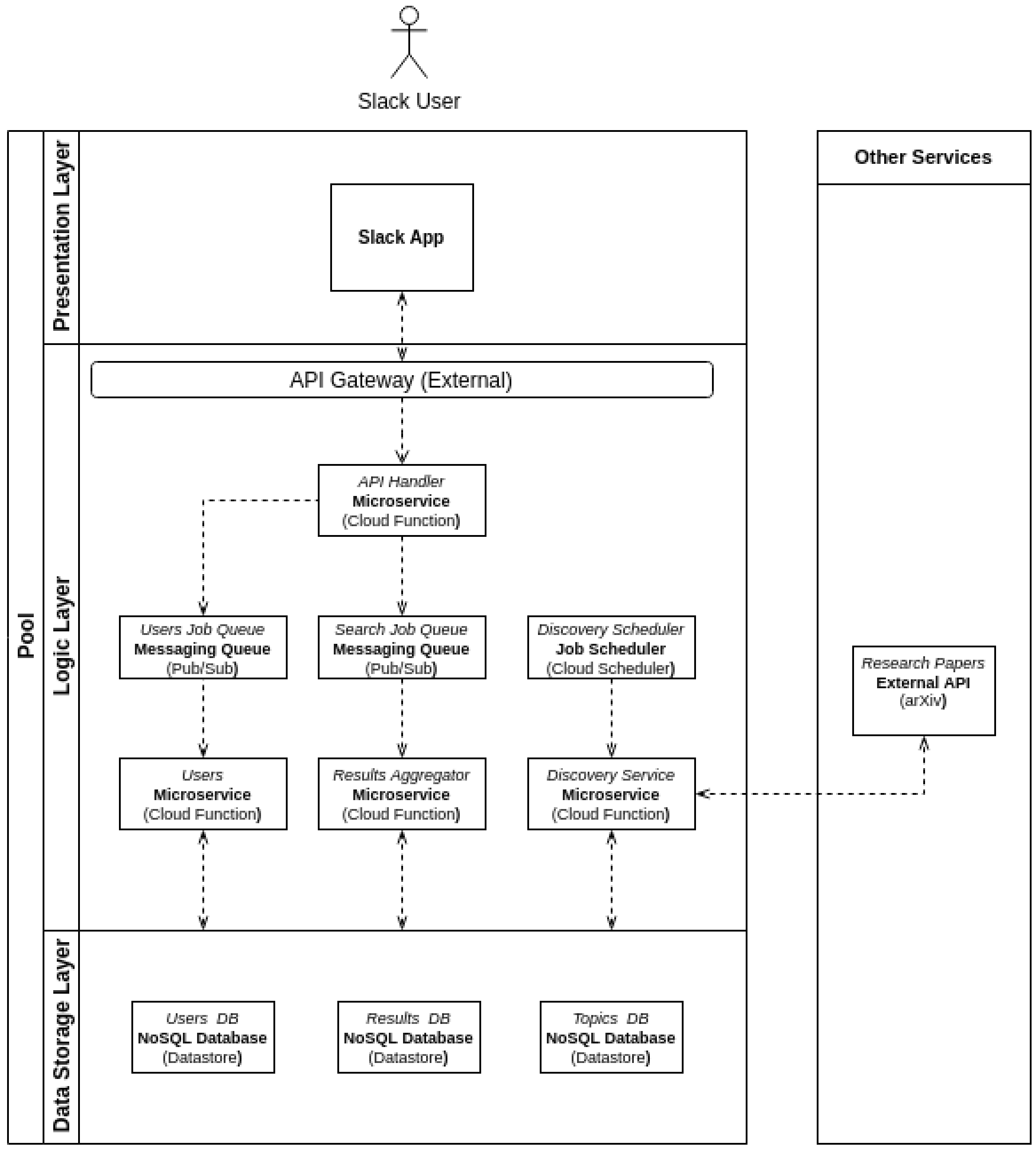
7. **Logging - Cloud Logging:** Cloud Functions have a strong integration with Cloud Logging and thus in order to fully leverage this, all we had to do was ensure we appropriately handled error scenarios using good coding practices and bubbled

up error messages along with the corresponding stacktrace. Debugging our application while deployed on GCP just involved inspecting the logs on the Cloud Console.

8. **Deployment - Goblet (3rd Party Tool):** For development and deployment we used Goblet which is a framework for writing serverless apis for GCP using python. It also provides support for local deployment which allowed us to debug using VS Code.

9. **Frontend - Slack App with a chatbot interface:** We leveraged Slack's Slash command functionality to offer a chat-like conversational interface through a Slack application. This made the tool behave like a 'research-assistant'.

ARCHITECTURE DIAGRAM



INTERACTION OF THE SOFTWARE AND HARDWARE COMPONENTS

The user interface for this application is a chat-like interface facilitated through a Slack application that needs to be installed in a Slack workspace. We make use of Slack's Slash commands which are messages that begin with / and trigger an HTTP request to a web service that can in turn post one or more messages in response. The application supports two Slash commands:

1. **/update-prefs** - This command is used by the user to enter (first interaction) or update their research area(s) of interests with the application. These preferences are stored against the user's Slack ID which is unique for all Slack workspaces (for the same email address). This commands a comma separated list of topics as an argument. Example usage: /update-prefs Edge Computing, Containerization

2. **/search-papers** - This command is used by the user to retrieve a curated list of 10 papers based on their area(s) of interests. If no arguments are passed to the command, then a search is performed on all topics found in the user's preferences. Alternatively, a comma separated list of topics can also be passed to the command to restrict the search to selected topics. By default the system only shows new results to the user if they were indexed after the user's last known search. Hence, if a user would like to see 'old' results, then they can simply run the command with the **-f** flag. Example usage: /search papers [-f] [comma-separated list of topics]

We make use of Google Cloud's API Gateway to expose two REST endpoints, one each for each Slash command:

1. /slash/prefs
2. /slash/search

The appropriate endpoint is called whenever a Slash command is executed. A dedicated API Handler microservice running on a Cloud Function backend is responsible for handling these calls. As we already know, the application primarily handles two kinds of user requests: a user preference insert/update request and a search request. Slash commands expect an acknowledgement response within 3 seconds of the request. Hence, all such requests are processed in an asynchronous manner in order to comply with the Slash command request contract.

In order to allow for such asynchronous request handling, we make use of message queues. More specifically, we make use of Google Cloud's Pub/Sub offering. Our system has two Pub/Sub queues:

1. Users Job Queue
2. Search Job Queue

Based on the request, the API Handler publishes a 'job' on the appropriate Pub/Sub queue. Then we have a couple of 'worker' services - Users and Results Aggregator that have a 'Push' subscription on the Users Job queue and the Search Job queue respectively. The worker services are invoked by the Pub/Sub to process the user request. Once the worker service finishes processing the request it sends back an appropriate response based on the nature of the request to the user through a callback URL that Slacks provides us in the payload of the original request.

Next we come to the Discovery Service that is responsible for integrating with external datasources for research papers. For now we have tested the integration with arXiv as the primary data source, however given our design it will be fairly easy to add support for other sources. The service runs a search against all topics listed in the Topics DB (Google Datastore) and updates the Results DB (Google Datastore) for any new results. We use a Cron Job Scheduler (Cloud Scheduler) to automatically run the Discovery Service at an interval of 24 hours. Results are timestamped to ensure that users are not served any stale results. The scheduler invokes the Discovery Dispatcher that parses all of the topics in the Topics DB and queues jobs for each topics in the Discovery Job Queue. The job queue then subsequently invokes the discovery service to retrieve papers from the web. We adopt this design to allow for parallel search across multiple topics.

Note that if a user enters a topic that doesn't exist in the Topics DB, then the Users microservice will add the topic to the Topics DB as well as queue a job in the Discovery Queue so that results are seeded for the new topic.

The user information including their preferences are stored in the Users DB (Google Datastore) and is accessed by both the Users and the Results Aggregator service. The Results aggregator service also interacts with the Results DB in order to curate a list of papers in order to serve a search request.

4. DEBUGGING AND TESTING

a. **Debugging:**

   It consisted of the following tools and the methods:

   i. **Use of Goblet**:
      Goblet is a framework for writing serverless APIs for GCP using Python. It is used for development as well as deployment. It provided support for local deployment which allowed debugging using VS code.

   ii. **Error Handling and Logging**:
      Errors are bubbled up appropriately along with error messages and stacktrace from every method. Google Cloud provides tight-knit integration of cloud functions with cloud logging which is used to view and debug application issues during development as well as deployment. Also, Goblet provides classes and methods for logging which is efficient for local debugging too.

b. **Testing:**

   i. **Component Testing**:
      Automated API tests are written to do component testing. Goblet's local deployment is also used during the component testing for local API testing.

   ii. **Integration Testing**:
      Manual testing is used majorly to do integration testing due to lack of control over slack based UI. Moreover, Google Cloud info logs are helpful to provide details of end to end working.

5. CAPABILITIES AND LIMITATIONS

a. **Capabilities:**

   i. **Design incorporating scale:**
      System is designed to handle a large number of users and to process large amounts of data. Data aggregation for topics is scaled using the divide and conquer approach to fetch the data in parallel from the external APIs. Data aggregation for each topic happens independently in parallel helping us scale the system. Pub-sub is used to handle the peak load as it can scale automatically as the number of users increases.

Also, Maximum connection limit on datastore is handled by throttling the maximum number of invocations of cloud functions. Retry mechanism with exponential backoff is added in cloud functions to handle transient failures.

ii. **Use of Goblet**: Goblet makes it easy and seamless to add additional APIs and tests using local deployment. It provides easy integration and support for API, pub-sub, schedule , cloud functions, cloud run etc.

b. Limitations:

i. **External API Limits:**
External APIs are rate limiting which limit our uses during the aggregation of data. It increases the latency and also creates scaling constraints on our system.

ii. **Result Aggregation and Format:**
Results returned to the slack interface need further polishing to have aggregation and sorting implemented in place. It also needs a recommendation system to be used for sorting according to the relevance of the content.

iii. **Possible format mismatches on Topic addition:**
Topic addition is a feature open to the users and it is prone to mismatches, errors and invalid formatting which needs to be corrected automatically through UI so that only the right details are passed along and used by the system for searching.

iv. **Limited UI interface capabilities:**
Slack is used as an interface for the users to interact. It has great UI features yet it does not come out of the box to handle all the integrations with the server and to provide flexibility in designing complex features on UI.